Contents

Chapter 1

Related work

1.1 Introduction

Self localization of robot inside a unknow environment passes through the extraction of relevant patterns from data acquired by sensors on-board. Such patterns, usually called *features*, must be chosen carefully and tested, in order to realize whether their are suitable for the task to be accomplished.
In this chapter, we will describe a procedure to estimate how much a certain features is useful, by exploiting the concept of *dissimilarity*, i.e. a measure of how much a sensor's perception (an image taken by a camera, for instance) changes with respect to a reference one.

The same importance must be given to the way we choose to represent a feature, in order to be able to individuate it when it figures out in different scans of robot. In this thesis, we decided to use lines as features; we will describe, then, in the beginning of this chapter, how the polar representation of a line was exploited in Hough domain, to individuate an algorithm able to solve the so called *scan matching* problem.
We will introduce, moreover, an other procedure that takes advantage of the geometric structure of environment and updates a list of plausible matches between lines in different scans. The output of procedure is melted with data from odometry, in order to update current robot's pose, through the use of Kalman filters.

1.2 Related work in scan matching

Given two sets of 2D data, the *scan matching* problem consists in finding a traslation T and a rotation R_ϕ that maximize the overlapping between the two sets. The scan matching algorithms usually work using data furnished by 2D range sensors and differ in their behaviour depending on the availability of an initial guess of solution.
A further classification of scan matching algorithms can be made on the basis of their assumption about the presence of noise in sensor's measurements and of features in surrounding environment. Feature-based algorithms, in particular, gained a great success because of their computational effectiveness, but it must be

underlined that the extraction of features may produce a loss of information.

When ambient contains features that are invariant to rotation and traslation, they can be easily extracted from scans and can be used to find a solution in a linear time.
If the extraction of features is not simple because of the structure of environment, then it is possible to use algorithms belonging to ICP family, which base their operation on a two step procedure:

- find an initial heuristic set of correspondences between points in two scans
- find a roto-traslation that more or less satisfies the set of correspondences

The two steps are repeated until the error falls below a certain threshold. The convergence is possible if scans are produced in two robot's positions that are close to each other.

1.2.1 Scan matching using the Hough transform

In a 2009 article by Censi, Iocchi and Grisetti, the problem of scan matching is afforded by the use of *Hough transform (HT)* and is, then, moved to the Hough domain. If we have an *input space* $\mathcal{S}$, the HT maps an input $i(\mathbf{s})$ (with $\mathbf{s} \in \mathcal{S}$) to an output function $\mathrm{HT}\{i\}(\mathbf{p})$, with $\mathbf{p}$ belonging to a *parameter space* $\mathcal{P}$.
We can define the HT of input $i(\mathbf{s})$ as

$$\mathrm{HT}[\mathcal{F}, i](\mathbf{p}) = \int_{\mathcal{F}_\mathbf{p}} i(\mathbf{s}) d\mathbf{s} \tag{1.1}$$

A possible choice for $\mathcal{P}$ is the the set of representations of lines in $\mathbb{R}^2$. If we recall the polar representation of lines

$$x\cos\theta + y\sin\theta = \rho \tag{1.2}$$

in which θ expresses the direction of line's normal vectors and ρ its distance from origin (see Figure 1.1), we can select a family of sets to apply HT:

$$\mathcal{F}_{(\theta,\rho)} = \{(x,y) | x\cos\theta + \sin\theta = \rho\} \tag{1.3}$$

On the other side, the input space is composed by points $P = \{p_j\}$, as they are returned by a range sensor. So we have that

$$\mathbf{s} := (x,y) \in \mathcal{S} := \mathbb{R}^2 \tag{1.4}$$

and

$$i(\mathbf{s}) = \sum_j \delta(\mathbf{s} - \mathbf{p_j}) \tag{1.5}$$

where δ is Dirac's impulse distribution.

HT has two fundamental properties:

- $\mathrm{HT}(\theta,\rho)$ is 2π-periodic
- the two points (θ,ρ) and $(\theta+\pi,-\rho)$ in $\mathcal{P}$ represent the same line in $\mathbb{R}^2$

Let's suppose that $i(\mathbf{s})$ and $i'(\mathbf{s})$ are two inputs, such that

$$i'(\mathbf{s}) = i(R_\phi \cdot \mathbf{s} + T) \tag{1.6}$$

Then, within parameter space $\mathcal{P}$:

$$\mathrm{HT}'(\rho,\theta) = \mathrm{HT}(\theta+\phi, \rho + (\cos\theta \quad \sin\theta)T) \tag{1.7}$$

Computing the HT passes through a discrete approximation of it, called *Discrete Hough Transform (DHT)*; the comparison with simple HT is shown in Figure 1.2. DHT allows to redefine our family of sets as:

$$\mathcal{D}_{(\theta,\rho)} = \{(x,y) |\ \rho \leq x\cos\theta + y\sin\theta < \rho + \Delta\rho\} \tag{1.8}$$

For a complete definition of the scan matching algorithm based

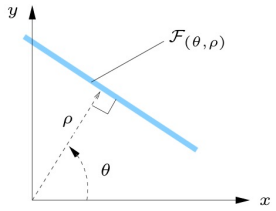


Figure 1.1: Hough transform

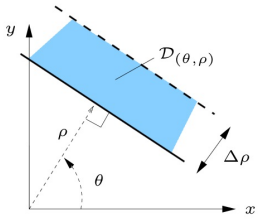


Figure 1.2: Discrete Hough transform

on HT, we need to define the so called *Hough spectrum*. For such purpose, we need a funcional g that is invariant to traslation. Theoretically speaking, this means that, given a function f:

$$f'(\tau) = f(\tau+\alpha) \Rightarrow g[f] = g[f'] \tag{1.9}$$

So, it is possible to define the Hough Spectrum as

$$\mathrm{HS}_g[i](\theta) := g[\mathrm{HT}[i](\theta,\cdot)] \tag{1.10}$$

An interesting property of spectrum is its *invariance*. If we define again two inputs $i(\mathbf{s})$ and $i'(\mathbf{s})$ as in (), the following statement is valid:

$$\mathrm{HS}_a[i](\theta) = \mathrm{HS}_a[i'](\theta+\phi). \tag{1.11}$$

This property, in case of the use of DHT in place of HT, holds only if the quantity $(\cos\theta \quad \sin\theta)T$ is a multiple of $\Delta\rho$.

The set of possible choices for g is wide and a natural one is the energy of the sequence in discrete domain:

$$g[f] = \sum_i f_i^2 \tag{1.12}$$

Since the property illustrated in () holds, it is possible to estimate ϕ by correlating the discrete spectra of sensor data $\mathrm{DHS}^S(\theta)$ and of reference data $\mathrm{DHS}^R(\theta)$. Then, local maxima of correlation are ordered; beginning with the greatest maximum, proceeding up to a chosen number, a new hypothesis for ϕ is formulated.
We can suppose that, if Φ is the domain in which we search for

the value of ϕ, such domain can have an upper and lower bound, in such a way that $\Phi = [-\phi_U, \phi_U]$. If this assumption cannot be made, then it can be simply set $\Phi = [-\pi, \pi]$.
The correlation between the two spectra can be obtained using the relation

$$\text{corr}_{DHS}(\phi) = \sum_{\theta \in \Theta} \text{DHS}^S(\theta) \cdot \text{DHS}^R(\theta - \phi) \tag{1.13}$$

The hypotheses $\phi_1, \ldots, \phi_k$ are retrieved from local maxima of correlation.

The scan matching problem is almost solved; what remains to be done is the estimation of traslation T. Let's assume $\phi = 0$ (this can be obtained by shifting the columns of DHT^S and DHT^R by the ϕ previously computed). We can choose an arbitrary $\theta = \hat{\theta}$. According to (), we can derive that

$$\text{DHT}^R(\hat{\theta}, \rho) = \text{DHT}^S(\hat{\theta}, \rho + (\cos\hat{\theta} \quad \sin\hat{\theta})T) \tag{1.14}$$

By correlating the columns of DHT^R and DHT^S, the projection of T onto the direction $\hat{\theta}$, $(\cos\hat{\theta} \quad \sin\hat{\theta})T = d(\hat{\theta})$, can be retrieved. Similarly to ϕ, a bound $[-|T|_U, |T|_U]$ for searching in the domain of $d(\hat{\theta})$ can be set.
Considering two different values for θ and the maximum of correlation, a linear system can be built to find T. Different pairs (θ_1, θ_2) furnish different results for T, that can be used as different hypotheses or combined to obtain an over-constrained system to be solved with least square estimation:

$$i = 1 \ldots n: \quad (\cos\theta_i \quad \sin\theta_i)T = d(\theta_1) \tag{1.15}$$

The elements of set $\{\theta_1, \ldots, \theta_n\}$ can be chosen among the maxima of sensor's DHS, i.e.

$$\{\theta_1, \ldots, \theta_n\} = \text{localMaxOf}\{\text{DHS}^S(\phi)\} \tag{1.16}$$

Finally, using the concepts described until this moment, it is possible to create a scan matching procedure, whose main steps can be summarized in the following:

- Compute DHT and DHS for reference and sensor data.
- Use local maxima of spectra cross-correlation to make hypotheses about ϕ.
- For each hypothesis produced at previous step, correlate columns of DHT to get linear constraints on T.
- Proceed following one of two possible paths:
  - combine linear constraints to get multiple hypotheses about T and order solutions on the basis of a likelihood function
  - produce a dense output by accumulating the results of correlation

1.2.2 Scan matching in polygonal environments

Many of the known matching methodologies require a good estimation of robot's initial pose which, when it is close to actual one, allows to limit the search space. The pose estimation and its update exploit a Gaussian model; thus, pose itself is accurately computed and updates can be calculated using Kalman filters. In particular, the extended Kalman filter models robot's pose as a Gaussian distribution $l(t) \sim N(\mu_l, \Sigma_l)$, using $\mu_l = (x, y, \alpha)^T$ as mean value and a 3×3 matrix for covariance Σ_L.
When the robot moves a distance δ and rotates by an angle θ, such motion can be described as $a \sim N((\delta, \theta)^T, \Sigma_a)$ and robot's pose is updated according to the following relations:

$$\begin{aligned} \mu_l &:= E(F(l,a)) = \begin{pmatrix} x + \delta\cos(\alpha) \\ y + \delta\sin(\alpha) \\ \alpha + \delta \end{pmatrix} \\ \Sigma_l &:= \nabla F_l \Sigma_l \nabla F_l^T + \nabla F_a \Sigma_a \nabla F_a^T \end{aligned} \tag{1.17}$$

where E is the expected value of function F and ∇F_l and ∇F_a are its Jacobians with respect to l and a.

From scan matching a pose update $s \sim N(\mu_s, \Sigma_s)$ is obtained, so the robot pose can be updated:

$$\begin{aligned} \mu_l &:= (\Sigma_l^{-1} + \Sigma_s^{-1})^{-1} \cdot (\Sigma_l'^{-1}\mu_l + \Sigma_s^{-1}\mu_s) \\ \Sigma_l &:= (\Sigma_l^{-1} + \Sigma_s^{-1})^{-1} \end{aligned} \tag{1.18}$$

Anyway, the success of Kalman filtering depends mainly on how the scan matching algorithm can estimate robot's pose. That is why, Gutmann, Weigel and Nebel, in 1999, proposed a new scan matching algorithm that exploits the polygonal structure of environment.
The procedure, known as *LineMatch*, is described by Algorithm 1. The main tasks performed by algorithm are extracting lines

Algorithm 1 LINEMATCH (M, S, P)

```
1:  Input : model lines M, scan lines S, pairs P
2:  Output : set of position hypotheses H
3:  if |P| = |S| then
4:      H := P
5:  else
6:      H := ∅
7:      s := SelectScanline(S, P)
8:     for all m ∈ M do
9:        if VerifyMatch(M, S, P ∪ {(m, s)}) then
10:           H := H ∪ {LINEMATCH(M, S, P ∪{(m, s)})}
11: return H
```

from current scan and comparing them with the ones contained in a *a priori* map. In particular, function *SelectScanline* picks a a new line to be matched, while *VerifyMatch* checks that a new pair *(m, s)* of matched lines is compatible with the pairs alreary accepted in previous iterations of procedure. Finally, the algorithm

returns a set of accepted pairs, that can be translated into positions from which robot may have taken scans (Figure 1.3). Proper constraints on lines' lengths and roto-traslations can allow algorithm to be more effective; this implies that it is not necessary to have a good estimate of robot's initial pose, since robot is able to recover from initial error. Once the matching activities are com-

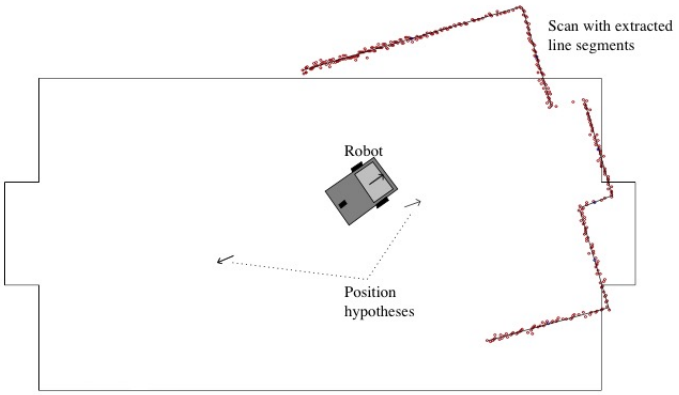


Figure 1.3: Example of two possible position hypotheses returned by LINEMATCH

pleted, the most plausible position is selected using odometry (a closest neighborhood policy will fit for the purpose) and is used to update robot's pose, by Kalman filters. A good choice for initial mean and covariance values may be

$$\mu_l := \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$$
$$\Sigma_l := \begin{pmatrix} \infty & 0 & 0 \\ 0 & \infty & 0 \\ 0 & 0 & \infty \end{pmatrix} \tag{1.19}$$

A summary of full algorithm is shown in Figure 1.4. The modular formulation supposes the presence of an *a priori* knowledge of the field in which robot is requested to operate.

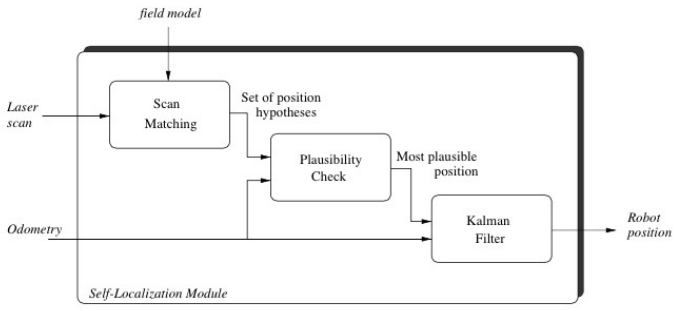


Figure 1.4: Modular representation of full algorithm

1.3 Selecting features to track

A scan matching algorithm, whatever its inner functionalities, can work properly only if the choice of features to extract has been made correctly, in order to maximize matching result.
Shi and Tomasi introduced, in 1994, a method for features selection based on the concept of *dissimilarity*, a measure of changes occurred to features between first frame and current one. When dissimilarity is too high, features should be ignored.

Considering frames acquired by a moving camera, all patterns that can be extracted vary in terms of intensity. Such changes are included in the definition of *motion*:

$$I(x,y,t+\tau) = I(x-\xi(x,y,t,\tau), y-\eta(x,y,t,\tau)). \tag{1.20}$$

What previous equation states is that an image taken at the instant of time $t+\tau$ can be obtained from one taken at time t by applying a motion, whose entity $\delta = (\xi, \eta)$ is called *displacement* of point $\boldsymbol{x}$= *(x, y)*.

If we speak in terms of windows, we can define the *affine motion field* as

$$\delta = D\boldsymbol{x} + d \tag{1.21}$$

with

$$D = \begin{pmatrix} d_{xx} & d_{xy} \\ d_{yx} & d_{yy} \end{pmatrix} \tag{1.22}$$

a deformation matrix; $\boldsymbol{d}$ is the traslation of a feature window's center.
By measuring the coordinates $\boldsymbol{x}$ with respect to the center of window, we get that a point $\boldsymbol{x}$ in the first image I moves to a point $A\boldsymbol{x}+d$ in second image J, with $A = \mathbf{1} + D$ and $\mathbf{1}$ a 2×2 identity matrix.
Thus, the tracking problem between two images I and J can be formulated in terms of finding the six parameters contained in D and $\boldsymbol{d}$. A model that uses small windows is preferable, since it causes less problems in terms of deph discontinuity; in such condition, the matrix D, that is harder to estimate because of the small variation of motion, can be assumed equal to zero, so

$$\delta = \boldsymbol{d}.$$

The equation () is not satisfied completely, then finding the six unknown motion parameters means to find A and $\boldsymbol{d}$ that minimize dissimilarity

$$\varepsilon = \int\int_W [J(A\boldsymbol{x}+\boldsymbol{d} - I(\boldsymbol{x})]^2 w(\boldsymbol{x}) d\boldsymbol{x} \tag{1.23}$$

with W the feature window and $w(\boldsymbol{x})$ a weighting function (in the simpleste case, it can be set to 1. In the other cases, it can be a Gaussian-like function).

When we have a simple traslation, matrix A must coincide with identity matrix. To minimize the value in equation (), it is possible to diffirentiate with respect to unknown parameters in D and $\boldsymbol{d}$ and

then set result to zero. The obtained system can linearized by a truncation in Taylor expansion:

$$J(A\boldsymbol{x}+\boldsymbol{d}) = J(\boldsymbol{x}) + g^T(\boldsymbol{u}). \tag{1.24}$$

This leads to a 6×6 system:

$$T\boldsymbol{z} = \boldsymbol{a} \tag{1.25}$$

with

$$\boldsymbol{z}^T = [d_{xx} \quad d_{yx} \quad d_{xy} \quad d_{yy} \quad d_x \quad d_y]$$

The system is completed by an error vector

$$\boldsymbol{a} = \int\int_W [I(\boldsymbol{x}) - J(\boldsymbol{x})] \begin{bmatrix} xg_x \\ xg_y \\ yg_x \\ yg_y \\ g_x \\ g_y \end{bmatrix} w\, d\boldsymbol{x} \tag{1.26}$$

and a 6×6 matrix

$$T = \int\int_W \begin{bmatrix} U & V \\ V^T & Z \end{bmatrix} w\, d\boldsymbol{x} \tag{1.27}$$

with

$$U = \begin{bmatrix} x^2g_x^2 & x^2g_xg_y & xyg_x^2 & xyg_xg_y \\ x^2g_xg_y & x^2g_y^2 & xyg_xg_y & xyg_y^2 \\ xyg_x^2 & xyg_xg_y & y^2g_x^2 & y^2g_xg_y \\ xyg_xg_y & xyg_y^2 & y^2g_xg_y & y^2g_y^2 \end{bmatrix}$$

$$V^T = \begin{bmatrix} xg_x^2 & xg_xg_y & yg_x^2 & yg_xg_y \\ xg_xg_y & xg_y^2 & yg_xg_y & yg_y^2 \end{bmatrix} \tag{1.28}$$

$$Z = \begin{bmatrix} g_x^2 & g_xg_y \\ g_xg_y & g_y^2 \end{bmatrix}$$

To be sure to have a good tracking, it is necessary to have a small motion between adjacent frames; this causes D to be small, too, so it can be set to zero. In this situation, determining deformation parameters can be pointless and dangerous as well, since it may lead to solutions with tiny displacements. Indeed, D and $\boldsymbol{d}$ are linked to each other through matrix V and an error in D produces an error in $\boldsymbol{d}$, too.
Then, when only $\boldsymbol{d}$ needs to be sought, it is sufficient to solve smaller system

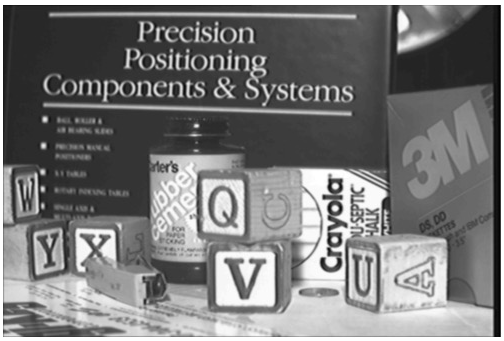
$$Z\boldsymbol{d} = \boldsymbol{e} \tag{1.29}$$

with $\boldsymbol{e}$ containing the last two elements in $\boldsymbol{a}$.
When the two frames to be compared are the first and current ones, a simple traslation model is not suitable and full system should be solved.

Considering a certain frame, not all its parts contain motion information and not all features are good for tracking, so a selection policy is necessary. A window can be tracked from frame to frame if the matrix Z is well-conditioned; thus, its eigenvalues must be large and cannot differ by too many orders of magnitude. Two small eigenvalues denote a constant intensity within window; a small and large eigenvalues indicate a unidirectional texture pattern; two large eigenvalues may correspond to corners and other easily trackable features.
In a few words, a window is accepted if, given the two eigenvalues λ_1 and λ_2 of Z we have

$$\min(\lambda_1, \lambda_2) > \lambda \tag{1.30}$$

with λ a chosen threshold.
An example of how textures may be extracted and accepted according to condition (1.30) is depicted in Figure 1.5.



(a)

(b)

Figure 1.5: Original image (a) and extracted features (b), using an acceptance criterion for windows, as explained in 1.30

Chapter 2

Basic concepts

2.1 Introduction

This chapter introduces to the main concepts on which this thesis is based.
First of all, we will concentrate ourselves on describing the concept of sensors and their importance in self-perception of robots. Sensors, indeed, are very useful to allow a robot to localize itself and have an idea of surrounding environment; anyway, to make all this possible, it is necessary to design them in such a way to satisfy a series of requirements. Sensors, although their immense utility, furnish noisy measurements with which localization algorithms must deal. In this chapter, we will describe one particular kind of sensors, the ones based on the *time of flight (TOF)* method.

The main tecnique used by robots to estimate their current position and orientation is *odometry*, which is based on data returned by robot's encoders. We will analyse the two types of errors it introduces and how they have been experimentally measured.

Data furnished by sensors is often huge, so it is important to extract only relevant parts of it, in order to reduce time in computation. We will describe how such parts can be extracted and which properties we expect them to have.

Finally, we will propose a process based on least squares minimization, that allows to estimate a robot's state (in terms of position and orientation), exploiting the error between measured and expected state value.

2.2 Robot sensors

Robots can be considered, among human artefacts, as the ones closest to the concept of a living being, with their capabilities to sense world and change their own choices on the basis of what they perceive in the surrounding environment. Without a sensing system, robots could be just able to repeat the same task and wouldn't be able to adapt their decisions to an environment continously changing around them. Thanks to sensors, robots can operate in places that are too dangerous for humans and interact with each other.

When designing a sensor, there is a small set of requirements

that should be kept into account:

- a good field of view, to satisfy the needings of the greatest number of applications
- a good minimum and maximum range of detection
- the largest accuracy and resolution
- a great ability to detect objects in the environment, dealing with phenomena such as reflection and noise due to ambient interference
- high update frequency, to have real time data at disposal as fast as possible
- generated data should be concise and easy to interpret
- low costs in designing, construction and maintenance
- low power consumption
- small size and weight

2.2.1 Range sensors

In many cases, a robot is required to build a map of its surrounding environment and for such goal a measurement of range between itself and obstacles is needed. There exists a class of range sensors widely used for such purpose based on the so called *time of flight (TOF)* method; an ultrasonic, RF or optical source generates an energy wave that propagates into ambient and the distance between robot and an object is computed as the product of the velocity of wave and the time required to travel the round-trip distance, according to relation

$$d = vt \tag{2.1}$$

where d is the round-trip distance, v is the speed of propagation of energy wave and t is the elapsed time. The time interval t in previous equations is the one that elapses while the energy wave reaches the obstacle and gets back to robot; thus, to retrieve the real range between robot and target, t must be divided by two.

The signal emitted by robot reaches an object and walks back on the same way to the robot, where it will be detected by a receiver, that can be located close to emitter or integrated with emitter itself. As it is evident in ( ), the range to an object can be easily retrieved and no further knowledge about object's nature is necessary.

Similarly to all kinds of sensors, even TOF systems are subject to measurements errors, which can be summarized in the following short list:

- **variations in the speed of propagation of emitted signal**

- **uncertainty in determining when reflected signal was detected**
  Such uncertainties depend on the fact that different surfaces reflect signal differently and this may cause time for detection of returned signal to rise. When a threshold on detection is established, more reflective objects are perceived as closer.
- **uncertanty in determining the exact round-trip time** t
  It is due to problems correlated to inner timing circuitry.
- **interaction between originated signal and surfaces**
  When source signal hits a surface, only part of it is reflected and perceived by the robot. The remaining part is propagated into ambient or passes through the hit object. Thus, it may happen that no signal is received by robot, particularly if emitting angle exceeds a certain threshold. The part of signal that is not sent back to robot, may be spread to envirnonment and perceived by another robot's sensors.

2.3 Odometry

Odometry is probably the most used method to estimate the position of a robot at any instant in time. The reasons of its success must be found in good accuracy, low cost and high sampling rates. Anyway, the way odometry computes robot's position (by integrating motion information across time) leads to errors; in particular, errors in estimating robot's orientation lead to large position errors, which grow proportionally with the distance afforded by robot.
Despite limitations due to errors, odometry gained a great success in mobile robots field, for at the least four reasons:

- other kinds of measurements can be itegrated to obtain a better accuracy
- in cases in which landmarks are used for helping to estimate robot's position, their number can be smaller, thanks to odometry's presence
- odometry allows to assume that robot is stuck in a certain pose for enough time to perceive all the landmarks in a given area and compute possible matchings with landmarks detected previously
- in some circumstances, because of the absence of external references (e.g. landmarks) or the impossibility to place any of them in a hostile environment, odometry can be the only way to estimate robot's position

Odometry exploits the measurements obtained from encoders mounted on robot's wheels to extimate its new pose. Let's suppose that, in a certain time interval, robot's wheels have pulses N_L and N_R. Let's denote with c_m the conversion factor that allows to

convert pulses into a linear displacement. We can suppose that

$$c_m = \frac{\pi D_n}{n C_e} \tag{2.2}$$

where:

- D_m is the nominal wheel's diameter
- C_e is the encoder resolution (pulses per revolution)
- n is the reduction gear ratio between motor and drive wheel

The incremental linear displacements $\Delta U_{L,i}$ and $\Delta U_{R,i}$ for the two wheels can be computed as:

$$\begin{aligned} \Delta U_{L,i} &= c_m N_L \\ \Delta U_{R,i} &= c_m N_R \end{aligned} \tag{2.3}$$

We can now compute the displacement for robot's centerpoint ΔU_i (that will be taken as a reference for robot's position) and change in orientation $\Delta\theta_i$:

$$\begin{aligned} \Delta U_i &= \frac{\Delta U_R + \Delta U_L}{2} \\ \Delta\theta_i &= \frac{\Delta U_R - \Delta U_L}{b} \end{aligned} \tag{2.4}$$

where b is the distance between the two points where wheels touch the ground.

Thus, the new orientation of robot at time i is:

$$\theta_i = \theta_{i-1} + \Delta\theta_i \tag{2.5}$$

The new position, at the same instant of time, of its centerpoint will be:

$$\begin{aligned} x_i &= x_{i-1} + \Delta U_i \cos\theta_i \\ y_i &= y_{i-1} + \Delta U_i \sin\theta_i \end{aligned} \tag{2.6}$$

2.3.1 Systematic and Non-Systematic errors in odometry

As explained previously, the odometry exploits three simple equations to estimate current position of robot. All of them, are based on the assumption that any encoder's measurement can be translated into a linear diplacement. Anyway, such assumption may lose its validity in certain circumstances; for instance, if a wheel slips, its encoder will register a measurement that is converted in a linear displacement, even if robot didn't move actually.
Slippage is only one of the possible causes of errors in odometry; they can me summarized and divided into the following two categories:

Systematic errors

- wheels with different diameters
- wheelbase different from nominal one

- wheels misalignment
- finite encoder resolution
- finite encoder sampling rate

Non-Systematic errors

- travelling on uneven floors
- travelling over objects
- slippage

Both errors categories are very difficult to manage; the first ones, in fact, are constantly present and accumulate over time; the latter, are unpredictable and robot must be able to react promptly when they manifest.

Across time, algorithms managing robot's position have thought it as surrounded by an error ellipse, denoting an area where robot may be in a certain moment. Ellipses grow with distance travelled, until an absolute measurement of position reduces ellipse's size. This method based on ellipses can be used to contain systematic errors only, since non systematic ones are unpredictable.

Estimating correctly the extent of odometric errors avoids further problems such as a wrong calibration of mobile platforms. In 1995, Borenstein and Feng developed a method based on a model which considers two erros to be dominant:

- the error due to different wheels' diameters $E_d = \frac{D_R}{D_L}$, with D_R and D_L, respectively, the actual diameters of right and left wheel
- the error due to uncertainty about real wheelbase $E_b = \frac{b_{actual}}{b_{nominal}}$, with b the wheelbase of robot

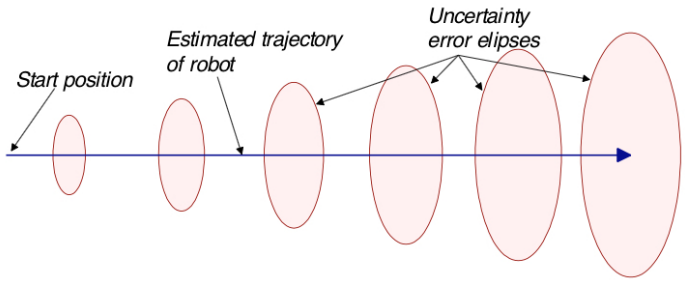


Figure 2.1: Ellipses, depicting robot's position error, grow as odometry uncertainty increases

2.3.2 Measuring Systematic errors

Before describing Borenstein and Feng's method to estimate systematic errors, it is useful to analyse Borenstein and Koren's method (1987).

Let's suppose that robot starts its path from an initial position

$<x_0, y_0, \theta_0>$ and has to move on a 4x4 meter square path (Figure 2.2). The robot is programmed to return to its initial position at the end of the path, but because of systematic errors, this will not happen accurately and a series of errors will be accumulated (*return position errors*):

$$\begin{aligned} \varepsilon_x &= x_{abs} - x_{calc} \\ \varepsilon_y &= y_{abs} - y_{calc} \\ \varepsilon_\theta &= \theta_{abs} - \theta_{calc} \end{aligned} \tag{2.7}$$

We have denoted with the *abs* subscript the absolute position and orientation of the robot and with *calc* subscript the position and orientation of robot according to odometry.

This kind of experiment is not suitable for testing odometry for differential drive platforms. That is why Borenstein and Feng introduced their *bidirectional square-path* experiment. This time, the robot is required to walk the path both in clockwise and counterclockwise direction. The errors E_d and E_b could compensate each other when robot travelled on one direction only; now, travelling on both direction, the two errors sum up.

If we let the robot travel on both directions n times (usually, $n = 5$), at the end of each run robot will accumulate a pose error as explained in (2.7). We can compute the center of gravity of these errors according to the following relations:

$$\begin{aligned} x_{cg}^{cw} &= \frac{1}{n}\sum_{i=1}^{n} \varepsilon_{x,i}^{cw} \\ y_{cg}^{cw} &= \frac{1}{n}\sum_{i=1}^{n} \varepsilon_{y,i}^{cw} \\ x_{cg}^{ccw} &= \frac{1}{n}\sum_{i=1}^{n} \varepsilon_{x,i}^{ccw} \\ y_{cg}^{ccw} &= \frac{1}{n}\sum_{i=1}^{n} \varepsilon_{y,i}^{ccw} \end{aligned} \tag{2.8}$$

The two absolute offsets of centers of gravity from origin are:

$$\begin{aligned} r_{cg}^{cw} &= \sqrt{(x_{cg}^{cw})^2 + (y_{cg}^{cw})^2} \\ r_{cg}^{cw} &= \sqrt{(x_{cg}^{ccw})^2 + (y_{cg}^{ccw})^2} \end{aligned} \tag{2.9}$$

Finally, a measure of systematic odometric error can be obtained as

$$E_{sys} = \max(r_{cg}^{cw}, r_{cg}^{ccw}). \tag{2.10}$$

The orientation error ε_θ is not considered in E_{sys}, since each orientation error translates into a position error.

2.3.3 Measuring non-Systematic errors

The square path test that was used to estimate systematic error, can be implemented again for non-systematic case, adding to the path some artificial bumps. Since return errors are sensible to the

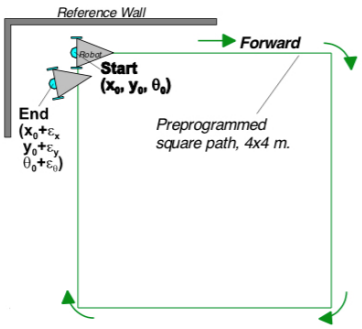


Figure 2.2: The 4x4 meter square path for Borenstein and Koren's method

positions of bumps, this time error ε_θ will be used. We can, then, compute an *average absolute orientation error*:

$$\varepsilon_{\theta,avg}^{nonsys} = \frac{1}{n}\sum_{i=1}^{n}|\varepsilon_{\theta,i,cw}^{nonsys} - \varepsilon_{\theta,i,cw}^{sys}| + \frac{1}{n}\sum_{i=1}^{n}|\varepsilon_{\theta,i,ccw}^{nonsys} - \varepsilon_{\theta,i,ccw}^{sys}| \quad (2.11)$$

The use of absolute values in previous equations is needed to avoid that two return orientation errors with opposite signs compensate each other. Thus, if after first run we have $\varepsilon_\theta = 1^\circ$ and after second one we have $\varepsilon_\theta = -1^\circ$, we will not erroneously derive that $\varepsilon_{avg}^{nonsys} = 0^\circ$.

2.4 Extracting features

In many different fields (machine learning and pattern recognition, just to mention a pair) the amount of data to manage (for example, the measurements returned by robot sensor) is too huge to be used in toto, so it becomes of great importance to extract only the necessary information, removing, moreover, all the redundancies that could make computation unfeasible. Thus., it is convenient to design processes that are able to extract information that is representative of data and allows to overcome the problem of using data in its totality. Such processes perform what is called *features extraction*, i.e. they retrieve (and sometimes compute) elements (*features*) that are distinctive of data, with which it is easier to operate; features are required to be informative, discriminative and indipendent. For some uses, features are even required to mantain some properties across space and time.
Usually, features have a numerical nature, but sometimes they can present a different aspect (in some cases they can be strings of characters or histograms).

Before designing an extraction process, features' structure needs to be planned, in order to respect the goal of good representation of data and to obtain elements with which working will be easy and effective. The choice about structure and computation of features depends on their final use and algorithms' purposes; here is a short list of common choices made in different fields:

- histograms of the distribution of black pixels in characters recognition

- phonemes in speech recognition
- repetitive words or text structures in spam detection, inside email inboxes
- edges and corners in computer vision

As explained in previous section, data retrieved from odometry are often inaccurate and they can lead to errors in robot's position estimation; features, after their extraction from sensors' data, can add a precious information allowing to reduce the errors and gain a better estimate of robot position in time. Indeed, each feature carries information about its reciprocal position with respect to robot and it can be used to improve localization methods
It is necessary to underline that even a well designed process of extraction can return a number of features that is too huge to be handled; so, it becomes crucial, in such cases, to be aware about which features are really necessary and which can be ignored. This mainly depends on the context in which robot operates, the perception system and the way features are represented.

2.4.1 Extracting lines as features

On deciding which features need to be extracted, geometric primitives are always a good solution to the problem; in particular, lines are the simplest choice, since most environments can be described using line segments. When receiving data from a ranger sensor (for instance, a laser range), one needs to operate with pairs (ρ_i, θ_i), which denote the position of i-th point in space, in polar coordinates, in robot's reference frame. Retrieving features from such information can be performed in two different ways:

- use a subset of points as features
- extract lines from the sequence of points, representing the part of environment seen by sensor during last scan

For the purpose of this thesis, we will consider the second option.

Across the years, many algorithms have been developed for extracting lines, starting from a laser's sequence of points, each with its pros and cons. The choice of an algorithm instead of another one depends mainly on one's needs. Sometimes speed needs to be preferred, in some other cases it is preferrable to give importance to quality, since bad extraction of features may lead to system's divergence. Complexity, too, is a particular not to be forgotten.
Here, we will describe the so-called *split and merge algorithm*, which operates using a *divide et impera* logic, starting with an initial set of points, progressively divided into subsets (on the basis of a check condition), until satisfying the conditions for determining the existence of a line (the subset under inspection has not enough points to be divided into two subsets and/or no point satisfies check condition to proceed to another division). The main steps of procedure are described by Algorithm 1. In particular:

- *isThereASetToSplit*($\mathcal{L}$) checks whether there is still a subset of points, among the ones in $\mathcal{L}$, that may be splitted into two subsets
- *takeASetToSplit*($\mathcal{L}$) extracts a set s_i among the candidates to be splitted in $\mathcal{L}$
- *fitALine*(s_i) returns a segment l_i that best fits the points in s_i (a simple choice may be computing the segment connecting the first and last point in s_i)
- *findFarthestPoint*(l_i, s_i) returns the furthest point P from l_i, among the ones in s_i
- *split*(s_i, P) splits the set of points s_i into two subsets: the first set contains the point preceeding P, the second one contains all remaining points
- *mergeCollinearLines*($\mathcal{L}$) merges segments that are collinear (i. e. segments that are very close to each other and lie on the same line)

Algorithm 2 Split and merge algorithm

1: **Input**: a set of all points s_1, an initially empty list $\mathcal{L}$ and a distance threshold d_p
2: **Output**: extracted segments (represented by their two extremes)
3: $\mathcal{L} \cup s_1$
4: **while**(*isThereASetToSplit*($\mathcal{L}$))
5: s_i = *takeASetToSplit*($\mathcal{L}$)
6: l_i = *fitALine*(s_i)
7: P = *findFarthestPoint*(l_i, s_i)
8: **if**(*distance*(P, l_i) > d_p)
9: (s_j, s_k) = *split*(s_i, P)
10: $\mathcal{L} \leftarrow \mathcal{L} - \{s_i\}$
11: $\mathcal{L} \cup \{s_k, s_j\}$
12: *mergeCollinearLines*($\mathcal{L}$)

2.5 Least squares estimation

Let $\mathbf{x}$ be a vector variable indicating the state of the system and let's suppose to have a series of functions

$$\{f_i(\mathbf{x})\}_{i=1:n}$$

which, given $\mathbf{x}$, predict its measurement.
Let's suppose, moreover, to have a series of real measurements $\mathbf{z_i}$. The goal is to extimate the state $\mathbf{x}$ that best fits the measurements $\mathbf{z_{i:n}}$.
We can, then, determine an error between each real measurement and the relative predicted one:

$$\mathbf{e}_i = \mathbf{z}_i - f_i(\mathbf{x}) \tag{2.12}$$

This error is assumed to be zero mean and normally distributed with information matrix Ω_i.

If we compute the squared form of error, we notice that it is a scalar and depends only on the state of the system:

$$e_i(\mathbf{x}) = \mathbf{e}_i(\mathbf{x})^T \Omega_i \mathbf{e}_i(\mathbf{x}) \tag{2.13}$$

Then, the problem of finding the state that best fits the real measurements can be reduced to find the state $\mathbf{x}^*$ such that

$$\begin{aligned} \mathbf{x*} &= \underset{\mathbf{x}}{\operatorname{argmin}} \; F(\mathbf{x}) \\ &= \underset{\mathbf{x}}{\operatorname{argmin}} \sum_i e_i(\mathbf{x}) \\ &= \underset{\mathbf{x}}{\operatorname{argmin}} \sum_i \mathbf{e}_i^T(\mathbf{x}) \Omega_i \mathbf{e}_i(\mathbf{x}) \end{aligned} \tag{2.14}$$

A general solution to the problem is to derive the global error function and find its nulls, but this would result in a complex solution which doesn't admit closed forms. So, proceeding with numerical approaches is a more practical path.

If a good initial guess is at disposal and the measurement functions are smooth in the neighborhood of minima, we can solve problem using iterative (local) linearizations. The iterative process is composed of three steps:

- linearize the problem around current initial guess
- solve a linear system
- compute a set of increments to be applied to current estimate in order to get closer to minima

The first step is linearizing the problem and this can be done using a Taylor expansion:

$$\begin{aligned} \mathbf{e}_i(\mathbf{x}+\Delta\mathbf{x}) &\simeq \mathbf{e}_i(\mathbf{x}) + \frac{\partial \mathbf{e}_i}{\partial \mathbf{x}} \Delta\mathbf{x} \\ &= \mathbf{e}_i + \mathbf{J}_i \Delta\mathbf{x} \end{aligned} \tag{2.15}$$

We can now place the Taylor expansion in the squared error:

$$\begin{aligned} & e_i(\mathbf{x}+\Delta\mathbf{x}) \simeq \\ & \mathbf{e}_i^T \Omega_i \mathbf{e}_i + \mathbf{e}_i^T \Omega_i \mathbf{J}_i \Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{J}_i^T \Omega_i \mathbf{e}_i + \Delta\mathbf{x}^T \mathbf{J}_i^T \Omega_i \mathbf{J}_i \Delta\mathbf{x} \\ & = \underbrace{\mathbf{e}_i \Omega_i \mathbf{e}_i}_{c_i} + 2 \underbrace{\mathbf{e}_i^T \Omega_i \mathbf{J}_i}_{b_i^T} \Delta\mathbf{x} + \Delta\mathbf{x}^T \underbrace{\mathbf{J}_i^T \Omega_i \mathbf{J}_i}_{\mathbf{H}_i} \Delta\mathbf{x} \\ & = c_i + 2\mathbf{b}_i^T \Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{H}_i \Delta\mathbf{x} \end{aligned} \tag{2.16}$$

The global error is the sum of the squared errors, so, from (3) and (5), we have:

$$\begin{aligned} F(\mathbf{x}+\Delta\mathbf{x}) &\simeq \sum_i (c_i + 2\mathbf{b}_i^T \Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{H}_i \Delta\mathbf{x}) \\ &= \sum_i c_i + 2(\sum_i \mathbf{b}_i^T)\Delta\mathbf{x} + \Delta\mathbf{x}^T (\sum_i \mathbf{H}_i)\Delta\mathbf{x} \end{aligned} \tag{2.17}$$

where

$$\mathbf{b}^T = \sum_i \mathbf{e}_i^T \Omega_i \mathbf{J}_i \tag{2.18}$$

$$\mathbf{H} = \sum_i \mathbf{J}_i^T \Omega_i \mathbf{J}_i \tag{2.19}$$

The expression of global error we have just obtained is quadratic in $\Delta\mathbf{x}$ and can be minimised; for this purpose, we can compute its derivative with respect to $\Delta\mathbf{x}$ in the neighbourhood of current solution $\mathbf{x}$:

$$\frac{\partial F(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq 2\mathbf{b} + 2\mathbf{H}\Delta\mathbf{x} \tag{2.20}$$

with the optimal solution reached for

$$\Delta\mathbf{x}^* = -\mathbf{H}^{-1}\mathbf{b} \tag{2.21}$$

Thus, an iterative algorithm can be designed; it will execute the following steps:

- linearize the system around current guess $\mathbf{x}$ and compute $\mathbf{e}_i(\mathbf{x}+\Delta\mathbf{x})$ as described in () and ()
- compute $\mathbf{b}^T$ and $\mathbf{H}$ as described in () and ()
- solve the system to get a new optimal increment $\Delta\mathbf{x}^*$, as indicated in ()
- update the previous estimate: $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}^*$

Chapter 3

Fast Position Tracking: overall approach

3.1 Introduction

As stated in previous sections, the amount of information returned by sensors is often too huge to be handled; thus, it is opportune to extract only relevant part that, in the case of laser scans, is in the form of lines, expressed in their polar representation. The set of lines furnishes a glimpse of environment's aspect, through the use of a *polyline structure*, which reflects how ambient has been seen by robot.

Given the lines obtained from each scan, it becomes interesting to track how the same line moves across scans and to recognise the same line in two (or more) different robot's perceptions: such problem, can be afforded using a new representation for lines, based on their normals and midpoints; this new representation, coupled with a least square minimisation approach, allows to design an iterative process that corrects, progressively, the estimate of a transformation matrix between two scans. When the final transformation T has been returned, it can be used to map the lines of a scan into another and determine associations between the lines revealed as the same across the two perceptions.

3.2 Landmarks and features

As discussed in last chapters, once a robot has acquired information from a sensor (a laser scan, for instance), it is necessary to extract only the relevant part of data, in order to reduce complexity without affecting performances. Such relevant part is formed by *features*, elements that describe reliably the part of environment perceived by a sensor's acquirement. They may represent physical objects and in robotics are often called *landmarks*, to underline the fact that they are exploited by robot during navigation. Without losing generality, we can assume that robot is always able to measure the bearing and range of landmarks in its own reference frame (Figure 3.1). Sometimes, an extractor adds a *signature* to measurements, in terms of a numerical value that can represent, for instance, an average color. The presence of sig-

nature is not strictly required and all what will be exposed in the following can be reduced to the case in which no signature has been introduced.

If we denote with f an extractor function, the extracted features can be represented as $f(z_t)$; let's indicate with r, ϕ and s the range, the bearing and the signature. We can obtain, then, a feature vector as a collection of triplets:

$$f(z_t) = \{f_t^1, f_t^2, \dots\} = \left\{ \begin{pmatrix} r_t^1 \\ \phi_1^t \\ s_t^1 \end{pmatrix}, \begin{pmatrix} r_t^2 \\ \phi_2^t \\ s_t^2 \end{pmatrix}, \dots \right\} \tag{3.1}$$

It is commonly accepted that features are independent from each other; such independence can be formalised as

$$p(f(z_t) \,|\, x_t, m) = \prod_i p(r_t^i, \phi_t^i, s_t^i \,|\, x_t, m) \tag{3.2}$$

where m is the map computed up to time instant t and x_t is current robot's state.
Actually, independence between features can be guaranteed only if the noise on a measurement (r_t^i, ϕ_t^i, s_t^i) is independent from the noise on another measurement (r_t^j, ϕ_t^j, s_t^j); when independence can be assumed, each feature can be processed separately and it is possible to design more efficient algorithms to implement probabilistic measurement models.

For feature-based algorithms, the map m is composed by a collection of features $m = \{m_1, m_2, \dots\}$. To each feature on the map corresponds a *location coordinate*, whose components can be denoted by $(m_{i,x} \quad m_{i,y})$. Assuming that, at time t, the i-th feature corresponds the j-th landmark and that robot's state, at the same instant, is described by $x_t = (x \quad y \quad \theta)$, we can model noise on measurement:

$$\begin{pmatrix} r_t^i \\ \phi_t^i \\ s_t^i \end{pmatrix} = \begin{pmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \\ s_j \end{pmatrix} + \begin{pmatrix} \varepsilon_{\sigma_r^2} \\ \varepsilon_{\sigma_\phi^2} \\ \varepsilon_{\sigma_s^2} \end{pmatrix} \tag{3.3}$$

where $\varepsilon_{\sigma_r^2}$, $\varepsilon_{\sigma_\phi^2}$ and $\varepsilon_{\sigma_s^2}$ are zero-mean Gaussian error variables with variances σ_r^2, σ_ϕ^2 and σ_s^2, respectively.

3.2.1 Extraction and model of features

When working with 2D landmarks, line segments represent a good choice for determining a reliable set of features, since they are simple geometric primitives but, at the same time, furnish a good amount of information to be used during localisation phases. The easiest way to represent a segment is by its endpoints; in some circumstances (for instance, the cases in which a robot scans an environment in which different corridors share a wall), the endpoints' information may be misleading, but the distance that can be measured between them is, at the least, a lower bound on the actual length of the segment they delimit.

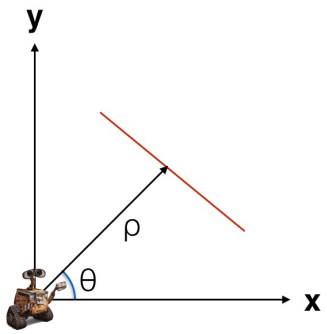


Figure 3.1: The range ρ and the bearing θ, relative to a landmark (line), expressed in robot's reference frame

Given a scan from laser, the *Split and merge* algorithm described in Section 2.4.1 can be used to extract segments, which can be characterised by their endpoints. When segments appear to be collinear and their extremes are close enough, they can be melted or joined, obtaining what is known as a *polyline structure*. Segments that could not be joined and whose length falls below a determined threshold, can be deleted, since they represent a useless source of noise. It is, moreover, possible to gain a polar representation of segments, using the aforementioned equation

$$x\cos\theta + y\sin\theta = \rho,$$

where ρ is the distance of line from the origin and θ is its orientation, in terms of its slope with respect to x axis. Both quantities are expressed in robot's reference frame.

3.3 Matching features

If we consider two scans i and j returned by robot's laser, according to previous section, we are able to extract lines from each of them, elaborate them and obtain a polyline structure formed by extracted segments.
What becomes of great interest at this moment is discovering whether one or more lines in scan i correspond to lines extracted from scan j. This is what is known as the *matching problem*.
Finding out the associations between lines belonging to different scans allows to compute a rigid transformation T from scan j to scan i; such transformation, is actually able to translate the reference frame of robot in which j was acquired onto reference frame in which i was expressed. The sought transformation should be able to maximize the overlapping between the two reference frames, minimising, at the same time, the value of an opportune error function.

3.3.1 Solving matching problem using least squares estimation

The problem of determining T can be formulated in terms of a least square estimation, by exploiting the concepts in Section 2.5. Let's suppose that a generic segment l_i in scan i can be expressed

as

$$l_i = \begin{pmatrix} p_i \\ n_i \end{pmatrix} \tag{3.4}$$

where p_i is a point on segment (middlepoint is the most convenient choice) and n_i is segment's normal, in its classical expression

$$n_i = (\cos\theta_i \quad \sin\theta_i)^T. \tag{3.5}$$

The needed transformation T will always have the form

$$T = \begin{pmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \tag{3.6}$$

where R is a 2×2 rotation matrix, $\mathbf{t}$ is traslation vector and $\mathbf{0}$ is a 1×2 zero vector. We can assume an initial value for T: in particular, T can assume the value of an initial guess or can be simply set to a 3×3 identity matrix. The transformation embodied by initial T can be applied to l_i, obtaining a new line l':

$$l' = Tl_i = \begin{pmatrix} Rp_i + t \\ Rn_i \end{pmatrix} \tag{3.7}$$

As it is evident from previous expression, to the generic point on segment a roto-traslation is applied, while only a rotation is applied to segment's normal. Supposing that new segment l' should be matched, after applying transformation, with a line $l_j = (p_j \quad n_j)^T$ in scan j, we can define an error function

$$e_{ij}(T) = \begin{pmatrix} Rp_i + \mathbf{t} - p_j \\ Rn_i - n_j \end{pmatrix} \tag{3.8}$$

A perturbation ΔT can be introduced to the error function, determining a new function

$$e_{ij}(T \oplus \Delta T) = \begin{pmatrix} \Delta R(Rp_i + \mathbf{t}) + \Delta T - p_j \\ \Delta R \cdot Rn_i - n_j \end{pmatrix} \tag{3.9}$$

Deriving previous expression, we get:

$$\frac{\partial e_{ij}(T \oplus \Delta T)}{\partial \Delta T} = \begin{pmatrix} \mathbf{I}_{2\times 2} & \begin{matrix} -p_{iy} \\ p_{ix} \end{matrix} \\ \mathbf{0}_{2\times 2} & \begin{matrix} -n_{iy} \\ n_{ix} \end{matrix} \end{pmatrix} = J_{ij}, \tag{3.10}$$

where subscripts ix and iy indicate the components x and y in p_i and n_i.

If we were able to rotate a single segment, in such a way its normal is parallel to x axis, we could obtain the useful result of having the segment's variance only moving along x. This result can be achieved by defining a covariance matrix as

$$\Omega_{ij} = \begin{pmatrix} \Omega_{p_i} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \alpha \mathbf{I}_{2\times 2} \end{pmatrix} \tag{3.11}$$

with

$$\Omega_{p_i} = R_i \begin{pmatrix} 1 & 0 \\ 0 & \varepsilon \end{pmatrix} R_i \tag{3.12}$$

and

$$R_i = \begin{pmatrix} n_{ix} & -n_{iy} \\ n_{iy} & n_{ix} \end{pmatrix}. \tag{3.13}$$

The constants α and ε are chosen in such a way they have small values.

Once covariance matrix Ω_{ij} has been computed, we can proceed to define an iterative process that permits to update the current value of transformation T. We first operate locally, according to equations

$$\begin{aligned} H_{ij} &= J_{ij}^T \Omega_{ij} J_{ij} \\ b_{ij} &= J_{ij}^T \Omega_{ij} e_{ij}. \end{aligned} \tag{3.14}$$

Then, we operate globally:

$$\begin{aligned} H &= \sum_{ij} H_{ij} \\ b &= \sum_{ij} b_{ij} \end{aligned} \tag{3.15}$$

Using global matrices, we can solve system

$$H \Delta T^\star = b \tag{3.16}$$

and obtain a new value T_{new} for the current estimate of transformation, according to

$$\begin{aligned} T_{\text{new}} &= \text{toMatrix}(\Delta T)^\star \cdot T \\ T &\leftarrow T_{\text{new}} \end{aligned} \tag{3.17}$$

where *toMatrix* is a function that builds a transformation matrix, starting from vector ΔT, containing displacements along axes and rotation value.

3.3.2 Determining associations between features

All what has been explained in previous section, can be inserted in an iteration process that tends to reduce progressively the error in alignment between two frames, while trying to find the transformation T to map one into the other.
Let's suppose to have two scans i and j and that we need to find the transformation T from j to i; let's suppose, moreover, to have chosen an initial value for T and a maximum number of iterations N. The steps of iteration process are summarised in the following.

Applying current transformation T. The first action is applying T to all the lines in j, in order to map them into reference frame i. As underlined previously, we can have an initial guess or an identity matrix as transformation, at first iteration of the process. Subsequently, we will apply, at iteration k, the transformation we obtained at $k-1$.

Data association. The lines that have just been transformed need to be associated to lines in i. First, we try to form associated

couples by comparing each transformed lined with the ones in i and finding the closest one comparing the normals. In particular, given the normals n_i and n_j of line l_i and transformed line l_j, respectively, we compute the dot product

$$p_{ij} = n_i^T n_j \tag{3.18}$$

Then, we add to a set the couple of associated indexes (i, j) if p_{ij} is greater than a properly chosen threshold. This gives us a set of putative associations that needs to be pruned.
Lately, for each couple of indexes, we evaluate the difference between the values of θ_i and θ_j of associated lines, along with the Euclidean distance between their midpoints. Both are requested two stay below two specific thresholds. All the couples of indexes, whose associated lines don't satisfy these last two conditions, are deleted from putative set.

Computing local and global matrices. Once we have a set of reliable couples of indexes (lines), for each couple we can compute the error e_{ij} as expressed in Equation (3.9); this allows us to determine the local matrices H_{ij} e b_{ij} (Equations from 3.9 to 3.14), for each couple of associated lines. Once all local matrices have been computed, we can derive global matrices H and b (Equations 3.15).

Solving linear system and updating current estimate of T. Once the global matrices have been computed, it is possible to prepare linear system in (3.16). Its resolution gives us the new estimate ΔT of transformation matrix between scans i and j. By applying new estimate to old one, we get a new transformation matrix that can be used for a new iteration of algorithm or can be directly returned if the maximum number of iterations has been reached. A block representation of a single iteration can be found in Figure 3.2.

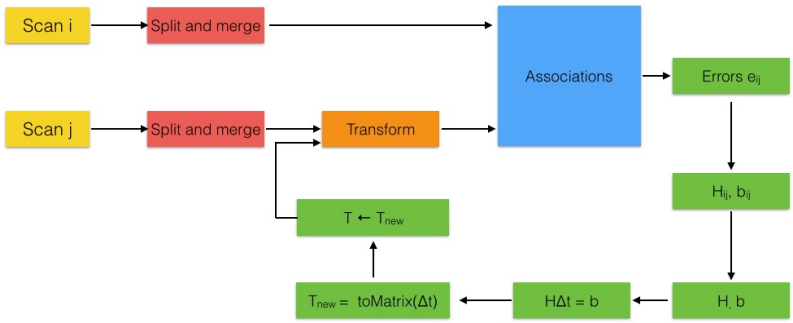


Figure 3.2: Block representation of a single iteration in T estimation

Chapter 4

Experiments

4.1 Introduction

This chapter deals with some tests designed to evaluate two of the parts composing thesis: the least square estimation (LSE) module and the one performing data association. For each test, results will be shown, along with input values given to each module. Before going into deep of tests, the reader will be introduced to the main means used for evaluation: the Robot Operating System and its embedded module *Stage*, used to generate data and simulations.

4.2 Robot Operating System

According to Willow Garage, the research laboratory that developed it in 2007, the **Robot Operating System** (**ROS**) can be defined as a *flexible framework for writing robot software*. Actually, ROS can be thought of as a collection of frameworks that can be exploited for the development of robot's software; such collection, furnishes all the typical functionalities of an operating system, along with robot-oriented capabilities. Here is a short list:

- hardware abstraction
- low-lovel device control
- exchange and recording (in *bag files*) of messages between processes
- robot-specific features: pose estimation, localisation, mapping, navigation (Figure 4.1)
- diagnostics, to produce, collect and aggregate data about robot, to control its state and produce an effective strategy to resolve incoming issues
- tools for debugging, plotting, and visualizing the state of the system under development (Figure 4.2)
- package management

Mainly, ROS' aim is simplifying the task of designing robot's behaviour across different platforms. It has been built in a collaborative mood, since it is open source and counts on the work of

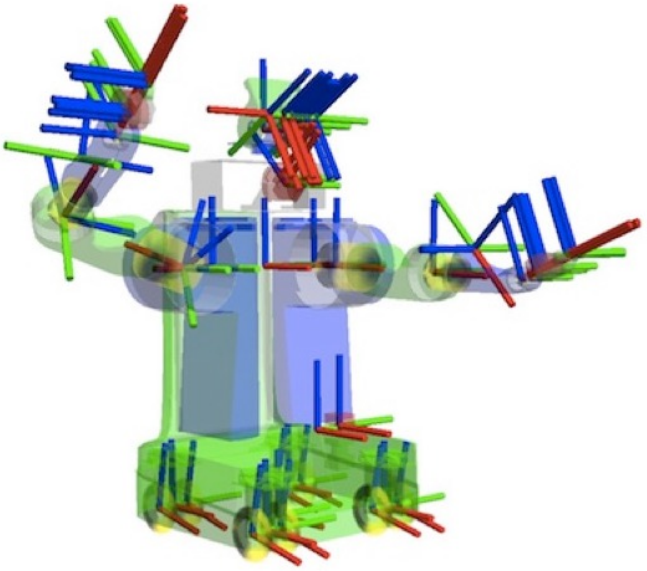


Figure 4.1: An example of robot-specific features: the *tf* library, which tracks the position of different elements of robot (sensors, links) in the robot's reference frame

different developers and institutions to enrich the basic functionalities.
ROS-based applications are organised in a graph structure composed of *nodes*, in which processes are executed and can receive or post messages regarding sensors, control, state, planning, actuators and so on. Even if a certain low latency is required in prevailing robotics, ROS is not a real time operating system, but it can be integrated with realtime code. There are, essentially, three kinds of softwares inside ROS:

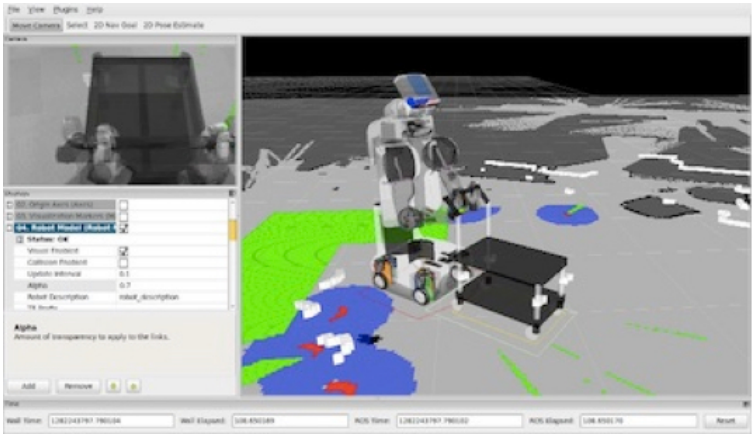
- language and platform independent tools for ROS distribution and building, under BSD licence[1]
- client library implementations (*roscpp, rospy, roslisp*), under BSD licence
- custom code using client libraries, which is sometimes commercial and is released under various open source licenses

The last category in previous list contains packages usually implemented for hardware drivers, robot models, perception, simultaneous localisation and mapping (SLAM).

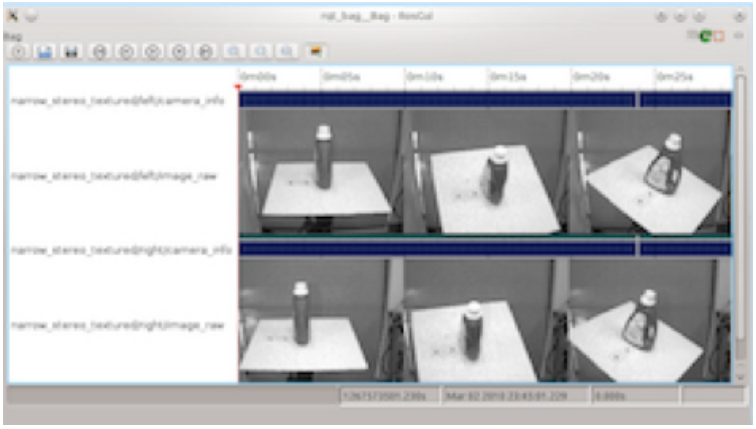
4.3 The ROS Stage Simulator

In order to test the line extraction, the least square estimation and the data association between lines in different scans, it is necessary to generate real data by simulating the robot behaviour (scanning comprised) in a realistic environment.

[1]The Berkeley Software Distribution (BSD) licence was originally created in 1990 to define the rules by which open source code can be used and distributed. The BSD licence requires the name of the code's developer to be always reported, even in documentation, along with a disclaimer specifying that code is released "as it is", without any garantee about its performance. The identities of contributors can't be used to endorse derived software, particularly for commercial purposes.



(a)



(b)

Figure 4.2: Two examples of ROS tools. The *rviz* tool (a) visualizes many common messages provided in ROS, such as laser scans, three-dimensional point clouds and camera images. It also uses information from the *tf* library to show sensor data in a chosen coordinate frame, along with a three-dimensional rendering of robot. The *rqt_bag* tool (b) records data to bags and can display their content.

When a real robot is not at the disposal, it is possible to simulate and make it move through the use of *Stage* simulator, which provides a virtual world populated with other mobile robots, sensors and objects that robot can sense and manipulate. *Stage* can be used as a stand-alone program or can be embedded inside a larger library or framework (we will use its embedded version inside a ROS node).
Stage itself provides the models of a series of actuators and sensors (sonar, infrared rangers, scanning laser rangefinder, bumpers, grippers); it provides even the model of a variety of mobile robot bases with odometric or global localisation.

To generate an accurate simulation in *Stage*, two elements are strictly necessary:

- a map (*world*) in which robot needs to move
- a player making robot move in simulated environment

With regards to maps, *Stage* furnishes a certain number of them that can be used to start learning how a simulation in its

environment works. Anyway, for more specific uses it could be required to create the map of a specific ambient in which robot has to move and sense. The definition of a map requires the creation of three objects:

- a *.png* file representing the shape of environment, created with the use of a laser rangefinder or by graphics editors (Figure 4.3)
- a *.world* file, containing the description of ambient (dimension of map, robots' initial positions and colors). An example relative to Figure 4.3 is showm in Figure 4.4
- a *.yaml* file, containing information, for instance, regarding the position of origin of axes on *.png* file, that will be used as a reference to track robot's position along its trajectory (Figure 4.5)

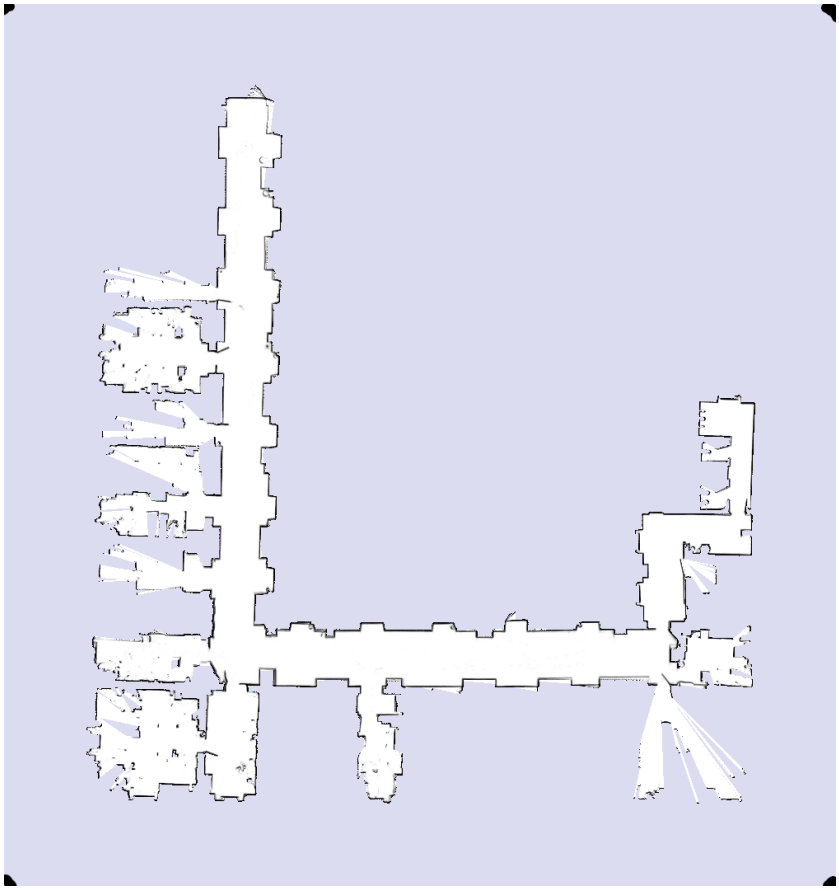


Figure 4.3: A *.png* file created for the definition of a map in *Stage*, depicting the first floor of DIAG department

Each robot on the map is usually represented as a small square with a different color; to make it move forward (along X or Y axes) or rotate, it is necessary to use another external library contained in another ROS node.
There is not a unique option, but a good choice is *teleop_base*; such node exploits the system for messages exchange described

```
define block model
(
  size [0.400 0.400 0.500]
  gui_nose 0
)

define floorplan model
(
  # sombre, sensible, artistic
  color "black"

  # most maps will need a bounding box
  boundary 1

  gui_nose 0
  gui_grid 0

  gui_outline 0
  gripper_return 0
  fiducial_return 0
  laser_return 1
)

# set the resolution of the underlying raytrace model in meters
resolution 0.02

interval_sim 100  # simulation timestep in milliseconds


window
(
  size [ 917.000 688.000 ]
  center [ -0.169 -0.592 ]
  rotate [ 0 0 ]
  scale 19.757
)

# load an environment bitmap
floorplan
(
  name "dis-B1"
  bitmap "dis-B1-2011-09-27.png"
  size [ 50.5 53.5 0.500 ]
  pose [ 12.8 14.5 0 0 ]
)

# robots
erratic( pose [   0.0   1.8 0 0 ] name "robot_0" color "blue")
#erratic( pose [  -1.0   1.8 0 0 ] name "robot_1" color "red")
#erratic( pose [  -8.0 -10.0 0 0 ] name "robot_2" color "green")
#erratic( pose [  -6.0 -10.0 0 0 ] name "robot_3" color "magenta")

#block( pose [ 8.0 2.0 0 0 ] color "red")
```

Figure 4.4: A *.world* file relative to Figure 4.3

```
image: dis-B1-2011-09-27.png
resolution: 0.05
origin: [-12.4537, -12.2613, .0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.2
```

Figure 4.5: A *.yaml* file relative to Figure 4.3

in previous section. *Teleop_base* receives commands from keyboard (Figure 4.6) and generates a message about which movement robot on map has to perform (either a straight progression or a rotation) and sends it to *Stage* simulator, which moves square associated to robot accordingly on the map.

```
Reading from keyboard
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

anything else : stop
---------------------------
```

Figure 4.6: Keyboard commands used by *teleop_base* controller

Stage itself is able to generate two kinds of messages:

- *geometry_msgs* which contain information about robot's position
- *nav_msgs* which contain information about data acquired by robot during its trajectory (usually in the form of couples (ρ, θ), describing the polar coordinates of points returned by range finder)

Such messages can be recorded in a *bag file*, a kind of container that intercepts and stores messages of specified types; recorded messages can be played back and received by new addresses. The message system requires a sort of DNS server that knows sender and receiver of each message; *roscore*, which can be executed from another ROS node, is the most natural choice for this purpose.

4.4 Results

In the following, it will be explained how some tests have been performed to evaluate the single parts forming the matter of this thesis; in particular, tests have been designed for the least squares and data association parts.

4.4.1 Preliminaries

All the testing activities that will follow, can be essentially divided into two parts:

- generation of testing data
- testing of overall approach

For the generation of data, we used the *Stage* simulator and the map of first floor of DIAG department, depicted in Figure 4.3. The robot, represented, as usually, as a small square, was made move

for a short journey along one of the corridors, using *teleop_base* and keyboard commands. While robot was moving, *Stage* generated *geometry_msgs* and *nav_msgs* messages, that were recorded in a bag file.
Subsequently, it was necessary to translate data in bag file into something more readable and useful for later uses; for such goal, an ad hoc ROS node was designed. Its task was listening to messages coming from the playback of bag file and record fundamental information on text files. In particular, the node extracted, from messages, information about odometric data and polar coordinates of scanned points, for each robot pose. The polar coordinates were converted in Cartesian coordinates (in robot's reference frame) before being saved on a text file.

The testing part was inserted in a separate ROS node; the node first loaded all the information previously recorded of text files, paying attention not to consider more than once the same pose and its associated data. From the points returned at each scan, lines were extracted using the *split and merge* algorithm described in Chapter 1.

4.4.2 Testing least squares estimation (LSE)

The first and easiest test that can be performed on LSE module requires to take the lines extracted from a scan of robot (at a given pose), to apply them a chosen transformation and to verify whether the LSE can reconstruct the applied transformation, without knowing it a priori. Let's start with a transformation that causes lines to translate, without introducing any rotation. For instance, let's consider the following matrix:

$$T_t = \begin{pmatrix} 1 & 0 & 5 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix}$$

We can easily compute its inverse:

$$T_t^{-1} = \begin{pmatrix} 1 & 0 & -5 \\ 0 & 1 & -7 \\ 0 & 0 & 1 \end{pmatrix}$$

We can apply T_t^{-1} to all the lines in a chosen robot's scan, run iteratively LSE module and check whether it is able to return T_t as the transformation to bring transformed lines to their original form. In Figure 4.7, original and transformed lines are displayed; in Figure 4.8 we have displayed even lines obtained by applying transformation returned by LSE to transformed lines. Obtained lines perfectly coincide with original ones, since LSE returns the correct transformation to bring lines back to their original form.

To make things a little bit complicated, let's add a rotation by $\theta = \pi/2$ to the simple translation given by T_t, gaining

$$T_{rt} = \begin{pmatrix} \cos\theta & -\sin\theta & 5 \\ \sin\theta & \cos\theta & 7 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 5 \\ 1 & 0 & 7 \\ 0 & 0 & 1 \end{pmatrix}$$

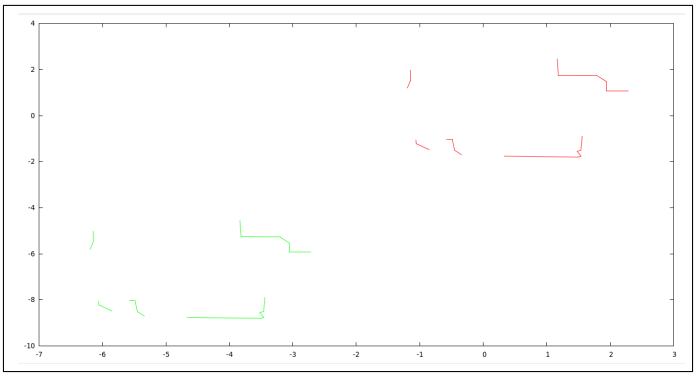


Figure 4.7: The original lines (red) and the ones obtained after applying T_t^{-1} (green)

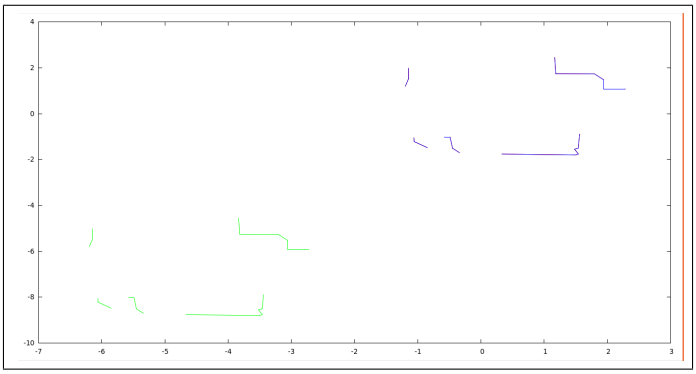


Figure 4.8: The original lines (red), the transformed ones obtained after applying T_t^{-1} (green) and the ones obtained by applying the transformation returned by LSE to transformed lines (blue). Blue and red lines coincide because LSE returns a transformation that perfectly allows to bring transformed lines to their original form

and its inverse

$$T_{rt}^{-1} = \begin{pmatrix} 0 & 1 & -7 \\ -1 & 0 & 5 \\ 0 & 0 & 1 \end{pmatrix}$$

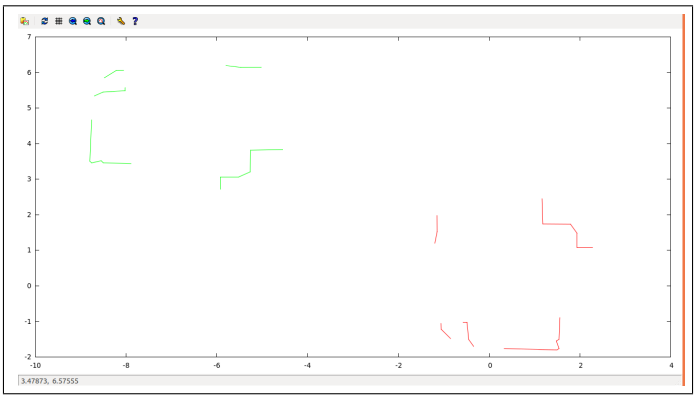


Figure 4.9: The original lines (red) and the ones obtained after applying T_{rt}^{-1} (green)

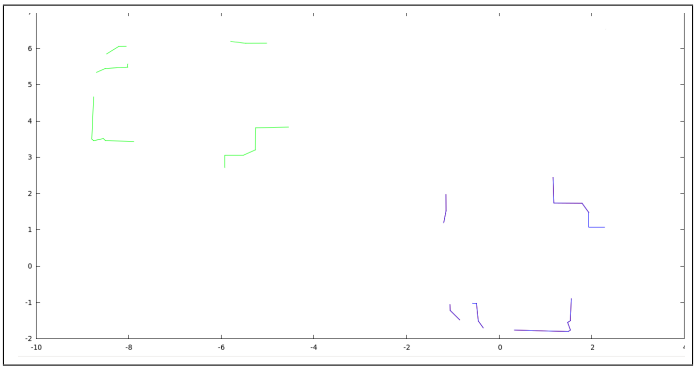


Figure 4.10: The original lines (red), the transformed ones obtained after applying T_{rt}^{-1} (green) and the ones obtained by applying the transformation returned by LSE to transformed lines (blue). Blue and red lines coincide because LSE returns a transformation that perfectly allows to bring transformed lines to their original form

As shown in Figures 4.9 and 4.10, even in this case the original lines (red) coincide with the ones obtained by applying to transformed lines the transformation returned by LSE. In this case, too, we have transformed original lines using T_{rt}^{-1} and asked LSE to guess that T_{rt} is the right matrix to bring transformed lines to their original form.

4.4.3 Testing data association

After verifying that simple tests executed in previous section gave good results, it was necessary to stress the system and prove its performance in a tougher case. Instead of creating a fictitious scan, whose extracted lines are obtained by applying an initial chosen transformation, let's consider two close robot poses p_1 and p_2, that differ in terms of odometric measurements. This means that, from these poses and scans acquired from them, we will certainly extract different lines or the same lines with different length

and slope.
We asked LSE to compute the transformation T_{21} that should map lines extracted in p_2's reference frame to p_1's. We then applied T_{21} to p_2's lines. The result is shown in Figure 4.11. As it is evident, LSE by itself is not able to furnish a good transformation to map correctly lines seen from p_2 to p_1's reference frame, since even lines that are evidently the same are far from each other and differently rotated.

So it is clear that LSE module needs to be integrated with some other sort of module to improve its performances; that is why, as shown in Figure 3.2, we coupled LSE with a data association module; the latter recalls iteratively LSE (which has an iterative nature of its own) correcting every time the set of initial couples with which LSE works. Such correction is made using last transformation T_{21} returned by LSE at last iteration (at first launch of LSE it is possible to use an identity matrix). We move lines seen from p_2 to p_1's frame and try to form couples of associations between the set of transformed lines and the ones in p_1's frame.
As explained in 3.3.2, a first set of putative couples is formed by comparing the normals of lines candidate to be associated: if the dot product exceeds a given threshold, the two lines form a temporarily accepted couple. For our test, we used a threshold $NORM\_THRESHOLD = 0.8$.

A putative couple is kept in set to be returned if the distance between midpoints of involved lines is lesser than $MP\_THRESHOLD = 0.1^2$ and the difference between their slopes falls below $ANGLE\_THRESHOLD = \frac{5\pi}{180}$.
A shown in Figure 4.12, now the full system composed by LSE and data association is able to return a transformation T_{21} that allows to optimally align p_1's and p_2's frames. In particular, Figure 4.13 shown the associations between lines obtained using the last T_{21} returned. It is evident that even the data association process benefits of the coupling of the modules.

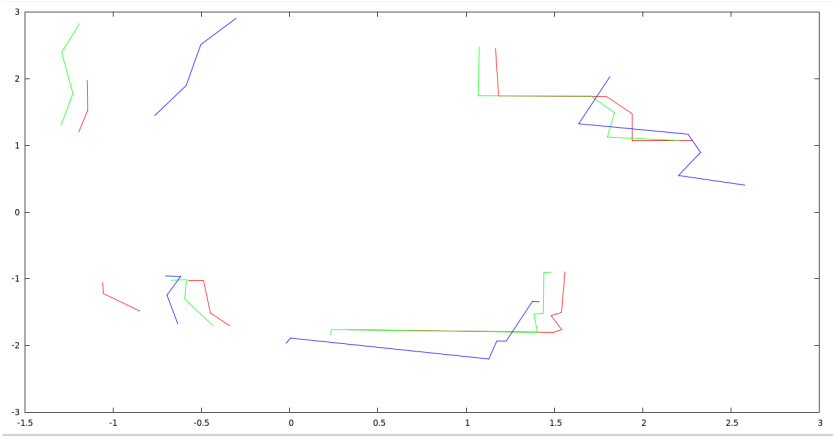


Figure 4.11: The lines extracted from p_1(red), the ones extracted from p_2 (green) and the ones obtained after applying T_{21} to lines in p_2 (blue)

A good question to answer is about a good number of iterations

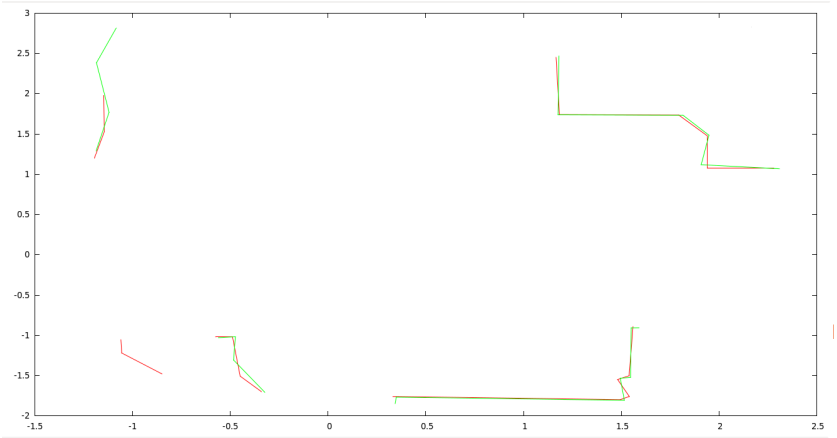


Figure 4.12: Lines acquired fron p_1 (red) and the ones seen from p_2 (green), after applying transformation T_{21} returned by the couples of modules made by LSE and data association

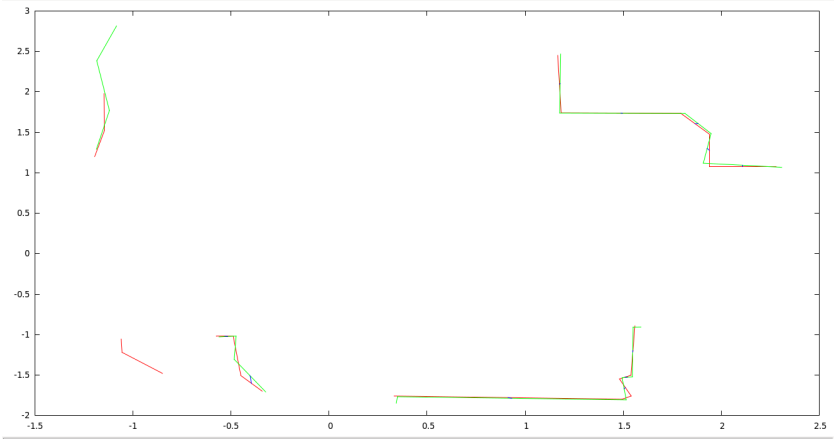


Figure 4.13: Associations between lines seen from p_1 (red) and p_2's transformed lines (green). Associations are shown in blue

for data association module, i.e. how many times LSE module has to be called and how many times its output needs to be corrected. As explained before, LSE works with an initial set of lines couples; for each of them, it is possible to calculate the error e_{ij} and covariance matrix Ω_{ij}, according to equations 3.8 and 3.11. So, for each couple, an error measure is given by

$$e_{ij}\Omega_{ij}e_{ij}^T \tag{4.1}$$

The quantities in (4.1) computed at i-th iteration of LSE can be summed, an error measurement χ_i. Thus, after N iterations, LSE return a total local error

$$\chi = \sum_i^N \chi_i \tag{4.2}$$

So, at j-th execution of data association module, we obtain a global error measurement

$$E_j = \frac{\chi_j}{a_j} \tag{4.3}$$

where a_j is the number of associations found at iteration j and χ_j is the local error of LSE module at the j-th iteration of data association module.
Figure 4.14 shows that, after remaining constant up to 50th iteration, error E_j almost falls down to zero and doesn't vary even after many other iterations. So, 50 proved to be a good choice when deciding how many times data association needs to be executed.

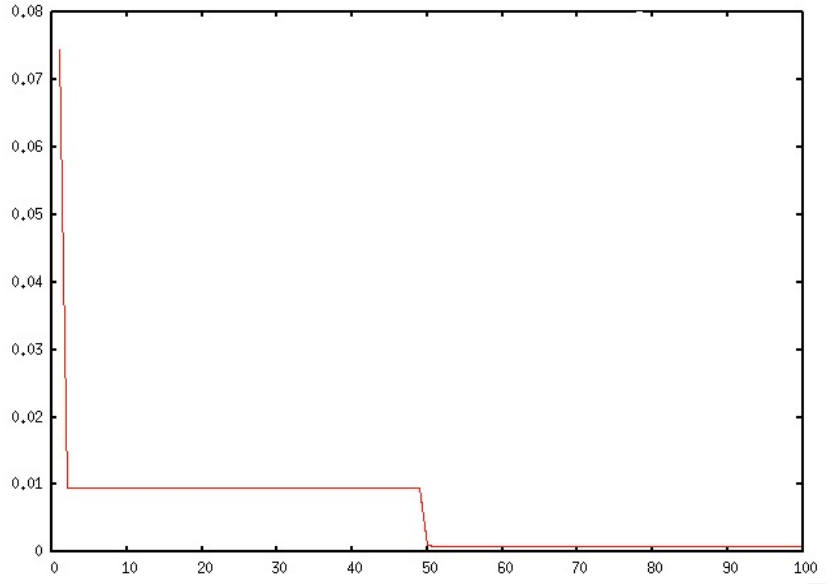


Figure 4.14: Number of iterations of data association module (X axis) and values of E_j (Y axis)

List of Figures